Koodipohjan jakaminen Windows 8- ja Windows Phone 8 -sovelluksissa

HAMK
HÄMEEN AMMATTIKORKEAKOULU

Ammattikorkeakoulun opinnäytetyö

Tietojenkäsittelyn ko

Visamäki, kevät 2013

Toni Ilomäki

TIIVISTELMÄ

HAMK
HÄMEEN AMMATTIKORKEAKOULU

HAMK Visamäki
Tietojenkäsittelyn ko
Systeemityö

Tekijä Toni Ilomäki **Vuosi** 2013

Työn nimi Koodipohjan jakaminen Windows 8- ja Windows Phone 8 -sovelluksissa

TIIVISTELMÄ

Työssä tutustutaan erilaisiin tekniikkoihin, joiden avulla voidaan hyödyntää mahdollisimman paljon samaa koodia Windows 8- ja Windows Phone 8 -alustoilla. Käytännön osuudessa on toteutettu kummallekin alustalle ristinolla-peli. Peli on verkossa pelattava kaksinpeli, josta löytyy myös keskustelutoiminto.

Peli toteutetaan kahden muun opiskelijan kanssa, ja he tekevät eri aihealueista omat opinnäytetyöt. Jarno Niemen työssä käsitellään Windows Communication Foundation -teknologiaa ja Azuren pilvipalvelua. Niko Kuusisen työ kertoo Windows Phone 8- ja Windows 8 -ohjelmistokehityksestä yleisellä tasolla. Työn toimeksiantajana toimii Hämeen ammattikorkeakoulun Tietojenkäsittelyn koulutusohjelma. Koulutusohjelmalle työstä toimitetaan toimiva ohjelmarunko ja dokumentti opetuskäyttöön.

Tässä työssä esitellään erilaisia tekniikoita, joiden avulla on helpompi jakaa ohjelmakoodia sovellusten välillä. Työn alussa jakotekniikat käydään yleisellä tasolla läpi ja niiden käyttöä tutkitaan esimerkkisovellusten avulla. Tämän jälkeen niiden käyttöä havainnollistetaan ristinolla-projektissa.

Työssä esiteltävä MVVM-malli on nykypäivänä hyvin yleisesti käytetty suunnittelumalli Microsoftin XAML-pohjaisissa teknologioissa. Sen avulla voidaan erottaa tehokkaasti näkymä sovelluslogiikasta. Portable Class Library mahdollistaa luokkakirjaston, joka on yhteensopiva kaikilla valituilla alustoilla. Windows Runtime Componentin avulla voidaan ajaa samalla tai eri ohjelmointikielellä tehtyä komponenttia erillisestä projektista. Näin on mahdollista tehdä laskentatehoa vaativat prosessit mm. C++:lla ja käyttää sitä eri kielellä. Jaetulla koodilla voidaan jakaa yksittäisiä luokkia sovellusten kesken. Käyttämällä yhteistä koodipohjaa eri alustoille toteutettujen sovellusten kesken saavutetaan ajallista ja rahallista hyötyä, kun tekeminen ja päivitykset kohdistuvat suoraan kaikille alustoille.

Avainsanat Windows 8, Windows Phone, MVVM, Windows Runtime, .NET

Sivut 33 s. + liitteet 6 s.

HAMK
UNIVERSITY OF APPLIED SCIENCES

HAMK Visamäki
Bachelor of Information Technology
Software Engineering

Author Toni Ilomäki **Year** 2013

Subject of Bachelor's thesis Code base sharing in Windows 8 and Windows Phone 8 applications

ABSTRACT

This thesis introduces different techniques that can be used to take maximum advantage of the same code in Windows 8 and Windows Phone 8 platforms. In the practical part tic-tac-toe game for both platforms has been implemented. The game is online two player playable, where is chat functionality.

The game is implemented with two other students and they do their theses with different subjects. Jarno Niemi's thesis introduces Windows Communication Foundation technology and the Azure cloud service. Niko Kuusinen's thesis introduces Windows Phone 8 and Windows 8 software development on a general level. The client in this thesis is HAMK University of Applied Sciences, Degree Programme in Business Information Technology. The degree program we deliver functional software trunk and document to teaching purposes. This thesis introduces different kind of technologies which will make it easier to share the software code between applications. At the beginning of thesis sharing techniques are presented on the general level. After this, the use of techniques is illustrated in the tic-tac-toe project.

The MVVM model presented in this thesis is nowadays a very commonly used design pattern in Microsoft's XAML-based technologies. It can be used to distinguish effectively the view from application logic. Portable Class Library enables class library that is compatible with selected platforms. Windows Runtime Component can be used to run a component, which is made of the same or different programming language. In this way it is possible to make the processes requiring computing power with for example C++ language and use with a different language. With the shared code single classes can be shared through applications. By using the common code base for applications with different platforms, temporal and financial benefits can be achieved when doing and updates are directly targeted to all platforms.

Keywords Windows 8, Windows Phone, MVVM, Windows Runtime, .NET

Pages 33 p. + appendices 6 p.

TERMIT JA LYHENTEET

.NET Framework on Microsoftin kehittämä kehitysalusta Windowsille, Windows Storelle, Windows Phonelle, Windows Serverille ja Windows Azurelle.

API (*Application Programming interface)* on ohjelmointirajapinta.

XAML (*Extensible Application Markup Language)* on XML-pohjainen kuvauskieli, jota käytetään pääsääntöisesti käyttöliittymien kuvaamiseen.

Depency Property on XAML-pohjaisissa komponenteissa käytettävä ominaisuus, joka tuo lisätoiminnallisuutta perinteisille CLR-ominaisuuksille.

CLR (*Commond Language Runtime*) on .NET Frameworkin ajonaikainen ympäristö, joka huolehtii mm. assemblyjen sijainneista, ja on vuorovaikutuksessa .NET perusluokka kirjastojen kanssa, hoitaa muistinhallinnan ja roskien keruun.

MSIL on paremmin tunnettu nimellä **CIL** (*Common Intermediate Language)*. CIL on ohjelmakieli, joka on .NET-kääntäjän alapuolella ja mm. C# ja VisualBasic käännetään kyseiselle kielelle ennen konekielistä käännöstä.

Hallittu koodi (*Managed code)* on koodia, jota ajetaan CLR:n alaisuudessa.

Natiivikoodi ***(****native code)* sanalla tässä tarkoitetaan ei-hallittua koodia. Esimerkiksi C ja C++, jolla ei ole mm. muistinhallintaa tai roskien keruuta

POCO (*Plain Old CLR Object*) on objekti, jolla on yleensä tila ja toimintoja, mutta joka ei sisällä riippuvuutta muihin sovelluskehyksiin vaan käyttää ainoastaan .NET:n perusluokkia.

SISÄLLYS

1 JOHDANTO

Työssä tutustutaan erilaisiin tekniikkoihin joiden avulla voidaan hyödyntää mahdollisimman paljon samaa koodia Windows 8- ja Windows Phone 8 -alustoilla. Käytännön osuudessa on toteutettu kummallekin alustalle kaikkien tuntema ristinolla-peli. Peli on verkossa pelattava kaksinpeli josta löytyy myös keskustelutoiminto. Peli on toteutettu yhteistyössä Jarno Niemen ja Niko Kuusisen kanssa. He kirjoittivat projektista omat opinnäytetyöt. Jarno Niemen työssä käsitellään Windows Communication Foundation -tekniikkaa ja Azuren pilvipalvelua. Niko Kuusisen työssä käsitellään Windows 8 ja Windows Phone 8 sovelluskehitystä yleisellä tasolla. Projekti toteutettiin täysin yhteistyössä, jolloin jokainen pääsi tutustumaan toisen aihealueeseen ja näin ollen oppimaan monesta aihealueesta.

Tässä opinnäytetyössä selvitetään, minkälaisia jakotekniikoita alustoille on ja kuinka niitä käytetään. Tekniikoiden yleisen läpikäynnin jälkeen, niiden yhteiskäyttöä tarkastellaan Ristinolla-pelin avulla. Työn lopussa on myös, miten tekniikoiden käyttöä voitaisiin vielä parantaa esimerkkiprojektissa.

Aihe on hyvin ajankohtainen johtuen Windows 8:n ja Windows Phone 8:n tuoreudesta. Moni toiselle alustalle tehdyistä ohjelmista kuitenkin julkaistaan mahdollisesti myös toisellekin alustalle. Tässä työssä esitellyillä tekniikoilla saman sovelluksen tekeminen kummallekin alustalle onnistuu mahdollisimman vähällä työllä.

Työssä ensimmäisenä esiteltävä MVVM-malli on myös muissa Microsoftin tekniikoissa hyvin yleisesti käytetty ja onkin nykyään Microsoftin suosittelema ohjelmiston suunnittelumalli. Esimerkiksi Windows Presentation Foundation (WPF) ja Silverlight -tekniikoissa tämä on erittäin paljon käytetty malli. MVVM-mallin perustuntemus on tärkeää nykyään, koska mm. monet Microsoftin tarjoamat ohje-esimerkit on toteutettu MVVM-mallia noudattaen. MVVM-mallia on alettu käyttämään muillakin ohjelmistoalustoilla. Esimerkiksi Javalle siitä löytyy jo omat kirjastonsa.

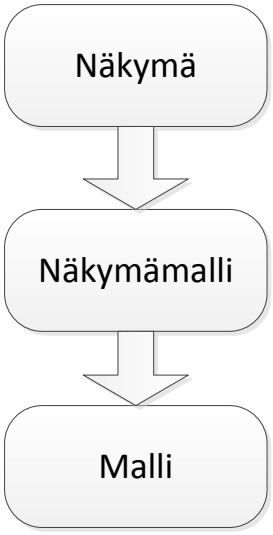
Muutkin työssä esiteltävät tekniikat ovat hyödyllisiä ajansäästön kannalta ja tämän kautta rahaa ohjelmistonkehityksessä, kun osa koodista voidaan kirjoittaa kerralla usealle alustalle. Jakotekniikat toimivat myös muilla alustoilla, kuten WPF, Silverlight ja Windows Phone 7.x pois lukien Windows Runtime Component, joka on ainoastaan Windows Phone 8:ssa ja Windows 8:ssa toimiva komponentti.

Projektin yhteistyökumppanina toimii Hämeen ammattikorkeakoulun Tietojenkäsittelyn koulutusohjelma. Koulutusohjelmalle toimitetaan työn lopputuloksena ohjelmarunko ja dokumentti, joita voidaan käyttää oppimateriaaleina.

2 MVVM-MALLI

Vuonna 2005 Windows Presentation Foundationin ja Silverlightin sovellusarkkitehti John Gossman esitteli blogissaan Malli-Näkymä-Näkymämallin (*Model-View-ViewModel*). Uuden WPF-teknologialle suunnatun suunnittelumallin, joka pohjautuu Martin Fowlerin vuotta aikaisempaan WPF-teknologialle suunnattuun PM (*Presentation Model*) -suunnittelumalliin. MVVM-mallissa on paljon samoja piirteitä kuin MVC:ssä (*Model-View-Controller).* Nykyään MVVM-malli on käytössä WPF:n lisäksi Silverlightissa, Windows Phonessa ja Windows 8:ssa. Kyseinen suunnittelumalli on nykyään Microsoftin suositus tehtäessä XAML-pohjaisia sovelluksia. (Garofalo, R. 2011, 1; Bugnion, L. 2012.)

MVVM-mallissa ohjelma jaetaan kolmeen osaan: Näkymä (*View),* Malli (*Model)* ja Näkymämalli (*ViewModel).* Malliin sijoitetaan järjestelmän tietomalli, joka voi sisältää esimerkiksi tarkistuksia ja muita oleellisia toimintoja kuten tallennuksen. Näkymä on käyttäjälle näkyvä sovelluksen käyttöliittymä, jossa käyttäjä voi suorittaa komentoja ja tarkastella tietoja. Näkymä on toteutettu XAML-tekniikkaa käyttäen ja parhaassa tapauksessa näkymän taustakooditiedosto (*code-behind file*) ei sisällä lainkaan koodia ja sen voi poistaa. Toisaalta tämä on monessa tapauksessa mahdotonta, koska mm. animaatioiden käyttö vaatii monesti kooditason muutoksia. Näkymämalli on MVVM-mallin oleellisin osa. Siellä sijaitsee näkymän logiikka, komennot ja se kapseloi mallin. Näkymämallilla ei ole tietoa tai riippuvuutta näkymästä. Kerrosten riippuvuussuhteet on esitetty kuvassa 1. Kuvasta nähdään, että ylempikerros tuntee ainoastaan kuvassa olevan alemman kerroksen (Smith, J. n.d. Advanced MVVM.)



Kuva 1 MVVM-mallin riippuvuussuhteet

2.1 MVVM-mallin perustekijät

2.1.1 Tiedon sitominen

Tiedon sitominen (*Data Binding)* on oleellisin osa käytettäessä MVVM-mallia. Sen avulla voidaan luoda sidos kahden objektin välille. Ensimmäinen osa sidoksessa on aina DepencyProperty-ominaisuus. Kyseinen ominaisuus löytyy lähes kaikista UIElement-luokasta periytetyistä luokista, kuten esimerkiksi TextBox, ListBox, ja lähes kaikista käyttöliittymän teossa tarvittavista luokista. Sidoksen toinen osa voi olla DepencyProperty-ominaisuus tai sitten perinteinen CLR-objekti. (MSDN. n.d. Data Binding Overview.)

Jos sidoksen asetukset on määritelty oikein, tieto haetaan ja päivitetään sidoksen päihin automaattisesti. Kun tekstikenttä on sidottu esimerkiksi perinteisen luokan ominaisuuteen, niin tekstikenttään tekstiä kirjoittaessa päivittyy se automaattisesti myös luokan ominaisuuteen. Tai jos ComboBox-kontrolliin on listattu taustavärejä ja ikkunan taustaväri on sidottu kyseisen listan valittuun väriin, niin vaihdettaessa väriä listassa, päivittyy automaattisesti ikkunan taustavärikin. (MSDN. n.d. Data Binding Overview.)

Sidos voidaan luoda joko XAMLia tai koodia käyttäen. On paljon helpompaa määrittää sidokset XAMLia käyttämällä, koska se tarjoaa yksinkertaisemman ja muistettavamman syntaksin sidokselle ja sen asetuksille. Sidos koostuu neljästä osasta: kohde (*target),* ominaisuus (*property),* lähde (*source*) ja polku (*path*). Kohde voi olla esimerkiksi TextBox-kontrolli, ominaisuus TextBox-kontrollin Text-ominaisuus, lähde CLR-objekti ja polku kyseisen objektin ominaisuus. Esimerkissä 1 ja 2 on toteutettu tämän esimerkin mukainen sidos. (MSDN. n.d. Data Binding Overview.)

Esimerkissä 1 on toteutettu Henkilo-luokan Etunimi-ominaisuuden sitominen TextBox-kontrollin Text-ominaisuuteen. Kuten esimerkistä 1 näkee, ensin on esitelty nimiavaruus, jonka avulla saadaan Henkilo-luokka käyttöön. Tämän jälkeen Henkilo-luokasta on luotu henkiloLuokka-niminen instanssi Grid-kontrollin resursseihin. Tämä instanssi asetetaan StackPanelin DataContext -ominaisuudelle, josta se periytyy sen sisällä oleville elementeille. Esimerkissä 2 on Henkilo-luokan toteutus.

```
<phone:PhoneApplicationPage

    xmlns:local="clr-namespace:MVVM_tietosidos_esimerkki"

    <!—..Koodia poistettu..-->
    <Grid>

        <Grid.Resources>
            <local:Henkilo x:Name="henkiloLuokka"/>
        </Grid.Resources>

        <Grid x:Name="ContentPanel">

            <StackPanel x:Name="stackpanel"
            DataContext="{StaticResource henkiloLuokka}">
            <TextBox x:Name="tekstikentta"
             Text="{Binding Path=Etunimi}"/>
            </StackPanel>
        </Grid>

    </Grid>

</phone:PhoneApplicationPage>
```

Lähde

Kohde

Ominaisuus

Polku

Esimerkki 1 Sidoksen määritys XAMLia käyttäen

```
public class Henkilo
    {
        private string _etunimi;

        public string Etunimi
        {
            get { return _etunimi; }
            set { _etunimi = value; }
        }
    }
```

Esimerkki 2 Sidoksessa käytetty Henkilo-luokka

Esimerkissä 3 on toteutettu sidoksen lisääminen TextBox-kontrollille kooditasolla. Kuten esimerkistä voi todeta, sidoksen tekeminen XAMLia käyttäen on huomattavasti selkeämpi ratkaisu. Silti sidoksen tekeminen koodilla on syytä opetella, koska ajonaikaiset sidokset täytyy toteuttaa kooditasolla.

```
Henkilo henk= new Henkilo();
Binding sidos = new Binding("Etunimi");
stackpanel.DataContext=henk;
tekstikentta.SetBinding(TextBox.TextProperty, sidos);
```

Esimerkki 3 Sidoksen toteutus kooditasolla

DataContext on kaikilla FrameworkElement-luokasta periytetyillä elementeillä. Tämän takia yleensä DataContext asetetaankin ikkunan tai sivun elementtiin, jonka jälkeen kaikilla on käytössään sama DataContext. Jos sidos

määritetään ilman lähdeasetusta, silloin käytetään automaattisesti ylemmän tason DataContext-ominaisuutta. Lähteen voi merkitä myös suoraan, mikä on käytännöllistä silloin, kun tarvitaan ainoastaan yhtä sidosta lähteestä. Jos lähde merkitään suoraan esimerkin 4 mukaan, ja DataContext on määritetty ylemmällä tasolla, niin lähempänä määritelty lähde ajaa aikaisemmin määritellyn lähteen ylitse. (MSDN. n.d. Data Binding Overview.)

```
<TextBox Text="{Binding
Source={StaticResource henkiloLuokka},Path=Etunimi}"/>
```

Esimerkki 4 Lähteen merkitseminen suoraan sidoksessa

Joskus tulee tilanne, etteivät sidoksen tietotyypit vastaa toisiaan. Ei voi esimerkiksi sitoa bool-tietotyyppiä System.Visibility -tyyppiin. Tällöin tietotyyppi pitää kääntää oikeaan muotoon tiedon siirtämisen aikana. Sidoksen asetuksiin voidaan määrittää konvertteri, joka hoitaa käännöksen. Konvertteri määritetään omaan luokkaansa ja sen pitää toteuttaa IValueConverter rajapinta, jossa on kaksi metodia: Convert ja ConvertBack. Convert-metodi hoitaa käännöksen, kun tieto tulee lähteestä kohteeseen ja ConvertBack-metodi kääntää takaisinpäin siirrettäessä. Esimerkissä 5 on toteutettu Convert-metodi ja sidoksen asetukset. ConvertBack-metodi on jätetty pois esimerkistä tämän ollessa täysin käänteinen Convert-metodin kanssa. (MSDN. n.d. Data Binding.)

```
public class BooleanToVisibilityConverter : IValueConverter
    {
        public object Convert(object value, System.Type targetType,
                              object parameter,
                              CultureInfo culture)
        {
         return (bool) value ? Visibility.Visible :
                               Visibility.Collapsed;
        }
    }
```

Esimerkki 5 bool-tietotyypistä Visibility-tyyppiin

```
  <Image Source="/polku" Visibility="{Binding
     boolTyyppi, Converter={StaticResource booleanToVisibility}}"/>
```

Esimerkki 6 Konvertterin käyttö XAMLissa

Tiedon sitomisesta voit lukea lisää MSDN:n sivulta:
http://msdn.microsoft.com/en-us/library/ms752347.aspx

2.1.2 INotifyPropertyChanged-rajapinta

Luvun 2.1.1 esimerkissä 1 tieto päivittyy käynnistyksen yhteydessä Henkilo-luokan Etunimi-ominaisuudesta näkymälle ja näkymässä tehdyt muutokset päivittyvät takaisin luokalle. Mutta jos jokin muu toiminto päivittää Etunimi-ominaisuutta suoraan luokassa, tämä muutos ei välity näkymälle. INotifyPropertyChanged-rajapintaa käytetään ilmoittamaan tietomallissa tapahtuvista muutoksista näkymälle. (Garofalo, R. 2011, 161–164.)

INotifyPropertyChanged-rajapinnassa on ainoastaan yksi toteutettava tapahtuma (*event*): PropertyChanged. Kun käytetään tiedonsitomista, käyttöliittymä seuraa tapahtuman laukeamista ja päivittää käyttöliittymää sen mukaan. Esimerkissä 6 on toteutettu Henkilo-luokkaan kyseinen rajapinta ja sen käyttö. Tämän muutoksen jälkeen esimerkissä 1 oleva tekstikenttä päivittyy, jos Etunimi-ominaisuutta päivitetään muualta kuin käyttöliittymästä. (Garofalo, R. 2011, 161–164.)

```
public class Henkilo : INotifyPropertyChanged
    {
        private string _etunimi;

        public string Etunimi
        {
            get { return _etunimi; }
            set
            {
                if(_etunimi==value)
                    return;

                _etunimi = value;
                RaisePropertyChanged("Etunimi");
            }
        }

        public event PropertyChangedEventHandler PropertyChanged;

        private void RaisePropertyChanged([CallerMemberName]
                                          String propertyName = "")
        {
            if (PropertyChanged != null)
            {
                PropertyChanged(this,new
                            PropertyChangedEventArgs(propertyName));
            }
        }
```

Esimerkki 6 Päivitetty Henkilo-luokka

2.1.3 ICommand-rajapinta

Perinteisessä sovelluksessa komennot sidotaan tapahtumankäsittelijöihin, kuten esimerkiksi napin Click-tapahtumaan ja sen toteutus sijoitetaan käyttöliittymän taustakooditiedostoon. MVVM-mallissa tämänkaltaista tilannetta tulisi kuitenkin välttää tapahtumienkäsittelijöiden ollessa hyvin käyttöliittymä riippuvaisia. MVVM-mallissa komennot sijoitetaan näkymämalliin, jolloin komennot ovat tietämättömiä, mistä komentokäsky tulee. Komennot sidotaan näkymään samalla lailla kuin ominaisuudetkin, mutta vain kontrollin Command-ominaisuuteen, joka aktivoituu hiiren painalluksesta tms. (Garofalo, R. 2011, 157–161.)

Command-ominaisuuteen sidottavan komennon pitää toteuttaa ICommand-rajapinta. ICommand-rajapinnassa on kaksi metodia: Execute, CanExecute ja yksi tapahtuma: CanExecuteChanged. Execute-metodi määrittää, mitä komento suorittaa, CanExecute palauttaa bool-tietotyypin ja sen avulla komento tietää, voiko Execute-metodia suorittaa ja CanExecuteChanged-tapahtuma laukaistaan, kun komennon tila on muuttunut. (MacDonald, M. 2010, 265–267.)

Esimerkissä 7 on näytetty, kuinka omaa ICommand-rajapinnan toteuttavaa luokkaa käytetään ja kuinka se sidotaan näkymämalli luokkaan. ICommand-rajapinnan toteuttava Komento-luokka löytyy liitteestä 1. Esimerkissä on painikekontrolliin sidottu NäytäViesti-ominaisuus. NäytäViesti-ominaisuus on määritelty esimerkin tapaan. Kun ohjelma avataan, sidottu komento tarkistaa VoikoSuorittaaKomentoa-metodin ja muuttaa sen perusteella painikkeen IsEnabled tilaa. Kun painiketta painetaan, tarkistetaan ensin VoikoSuorittaaKomentoa-metodin tulos. Jos metodi palauttaa tosiarvon, niin SuoritaKomento-metodi ajetaan.

```
private Komento _näytäViesti;

public Komento NäytäViesti
{
   get { return _näytäViesti ??
                        (_näytäViesti = new Komento(param =>
                         SuoritaKomento(), param =>
                         VoikoSuorittaaKomentoa()));
       }
}

private void SuoritaKomento()
{
   MessageBox.Show("Hello world");
}

private bool VoikoSuorittaaKomentoa()
{
   return true;
}
```

Esimerkki 7 Komennon käyttö

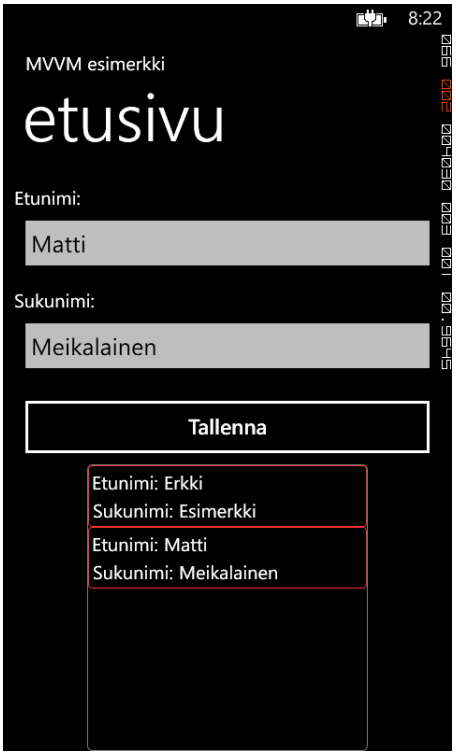
```
<Button Content="Näytä viesti" Command="{Binding NäytäViesti}"
Width="220" Height="90"/>
```

Esimerkki 8 NäytäViesti-ominaisuuden sitominen painikkeeseen XAMLissa

2.2 MVVM-mallin käyttö

MVVM-suunnittelumallia käytettäessä projekti kannattaa jakaa selvästi kolmeen osaan. Jakaminen toteutetaan omilla kansioilla tai luokkakirjastoilla. Se, kuinka sen tekee, riippuu projektista ja omista mieltymyksistä. Kuitenkin silloin, kun projektia on tekemässä useampi tai MVVM-mallin osia tullaan käyttämään muissakin sovelluksissa, niin kerrokset kannattaa jakaa omiin kirjastoihin.

Esimerkkinä on toteutettu pieni Windows Phone 8 -sovellus, jossa on kaksi tekstikenttää, painike ja lista. Käyttäjä voi syöttää kenttiin etunimen ja sukunimen, minkä jälkeen käyttäjä voi painaa Tallenna-painiketta. Painikkeen painamisen jälkeen henkilö lisätään alla olevaan listaan. Painiketta ei voi kuitenkaan painaa ennen kuin kummassakin kentässä on tekstiä. Kuvassa 2 on kuvakaappaus esimerkkiohjelmasta.



Kuva 2 Esimerkkisovellus käytössä

Esimerkkisovelluksen mallikerroksessa ei ole kuin yksi luokka: Henkilo (Liite 2); Henkilo-luokka on POCO, jossa on kaksi ominaisuutta: Etunimi ja Sukunimi. Näkymämallina on MainPageNäkymäMalli-luokka (Liite 3), jossa on sidottavat Etunimi-, Sukunimi-, HenkiloLista- ja TallennaKomento-ominaisuudet. Luokka myös toteuttaa INotifyPropertyChanged-rajapinnan. Näkymämalli noudattaa MVVM-mallin puristista linjaa toteuttamalla Henkilo-luokan näkymässä tarvittavat ominaisuudet uudelleen. Toinen (ja monesti helpompi) tapa on paljastaa Henkilo-luokan instanssi julkisessa näkymämallin ominaisuudessa ja sitoa kyseisen instanssin kautta Henkilo-luokasta tarvittavat ominaisuudet näkymään. Esimerkissä 9 on toteutettu tällainen sidos. Tämä kuitenkin paljastaa mallin toteutusta näkymälle. Kumpikaan tapa ei ole väärä ja kuten Garofalo (2011, 157) totesi: ”Rehellisesti sanottuna ei ole ”parasta” tapaa, on vain erilaisia tapoja samaan ongelmaan”.

```
<TextBlock Text="{Binding HenkiloLuokanInstanssi.Etunimi}"
```

Esimerkki 9 Näkymämallin Henkilo-luokan julkisen instanssin kautta sitominen

Esimerkissä 10 on näkymämallin TallennaHenkiloKomento- ja HenkiloLista-ominaisuuksien toteutus. HenkiloLista-ominaisuuden kokoelmana käytetään ObservableCollection<T>-luokkaa, koska luokka toteuttaa INotifyPropertyChanged- ja INotifyCollectionChanged-rajapinnat. Näin toteutettuna kokoelma osaa ilmoittaa lisäyksistä ja poistoista automaattisesti näkymälle. TallennaHenkiloKomento-ominaisuus on Komento-luokan instanssi (Liite 1). Se asettaa yksityiselle Henkilo-luokan instanssille etunimen ja sukunimen ja lisää henkilön tämän jälkeen listalle. TallennaHenkiloKomento-ominaisuudessa on myös tarkistus, jossa katsotaan, onko kumpikaan tekstikenttä tyhjä.

```
public ObservableCollection<Henkilo> HenkiloLista
    {
        get
            {
               return _henkiloLista ?? (_henkiloLista=
                            new ObservableCollection<Henkilo>());
            }
            set
            {
                if(_henkiloLista==value)
                    return;

                _henkiloLista = value;
                RaisePropertyChanged("HenkiloLista");
            }
        }

private Komento _tallennaHenkilo;

public Komento TallennaHenkiloKomento
    {
        get
           {
               return _tallennaHenkilo ??
                         (_tallennaHenkilo =
                          new Komento(suorita =>
                        {
                            _henkilo.Etunimi = _etunimi;
                            _henkilo.Sukunimi = _sukunimi;
                            _henkiloLista.Add(_henkilo);
                        },
                        voikoSuorittaa =>
                        {
                            if(!string.IsNullOrEmpty(_etunimi)
                             && !string.IsNullOrEmpty(_sukunimi))
                                return true;

                                return false;
                        }));
            }
   }
```

Esimerkki 10 Näkymämallin HenkilöLista- ja TallennaHenkiloKomento-ominaisuuksien toteutus

Näkymänä esimerkissä on Windows Phone 8 projektipohjan oletuksena tuleva MainPage-sivu (Liite 4). Näkymä on sidottu näkymämalliin sivun taustakoodi tiedostossa. Siellä on asetettu sivun DataContext-ominaisuuteen MainPageNäkymäMalli-luokan instanssi. Näin on saatu koko sivulle sama sidontalähde. Näkymämallin sidonnan näkymään voi tehdä myös muillakin tavoilla, kuten esimerkiksi suoraan XAML:ssa (ks. kappale 2.1.1), käyttämällä malleja kuten Inversion of Control, Depency Injection ja Service Locator. Tässä esimerkissä käytetyt luokat löytyvät kokonaisuudessaan liitteistä.

3 PORTABLE CLASS LIBRARY

Portable Class Library eli lyhennettynä PCL on Visual Studio 2010:neen lisäosana saatava ja Visual Studio 2012 versiossa valmiina oleva projektimalli. PCL:n avulla voidaan luoda kirjasto, joka kääntyy valituilla alustoilla suoraan, ilman tarvetta tehdä alustakohtaisia muutoksia. Käännettäessä PCL:ä MSIL muotoon PCL käännetään ainoastaan kerran, jolloin jokainen kirjastoa tarvitseva samalla koneella oleva sovellus voi käyttää sitä suoraan. (Trofin, M. 2012; MSDN. Cross-Platform Development with the .NET Framework. n.d.)

PCL mahdollistaa esimerkiksi pelin logiikan jakamisen, jolloin ei ole tarvetta kopioida sitä eri projekteihin. Näin ollen koodin hallittavuus paranee ja aikaa säästyy huomattavasti, kun ei ole tarvetta ylläpitää samaa koodia jokaiselle projektille erikseen. Kääntäjä pitää huolen, että kaikki kirjaston koodit ovat yhteensopivia valituilla alustoilla; käyttäjä ei voi vahingossakaan käyttää PCL:ssä sellaista luokkaa mitä valitut alustat eivät tue. Ennen PCL:n tuloa piti koodia jakaa eri alustoiden kesken kopioimalla ja käyttämällä ehdollisia kääntäjän vihjeitä (*Conditional compiler directives*). (Trofin, M. 2012; MSDN. Cross-Platform Development with the .NET Framework. n.d.)

3.1 Tuetut ominaisuudet

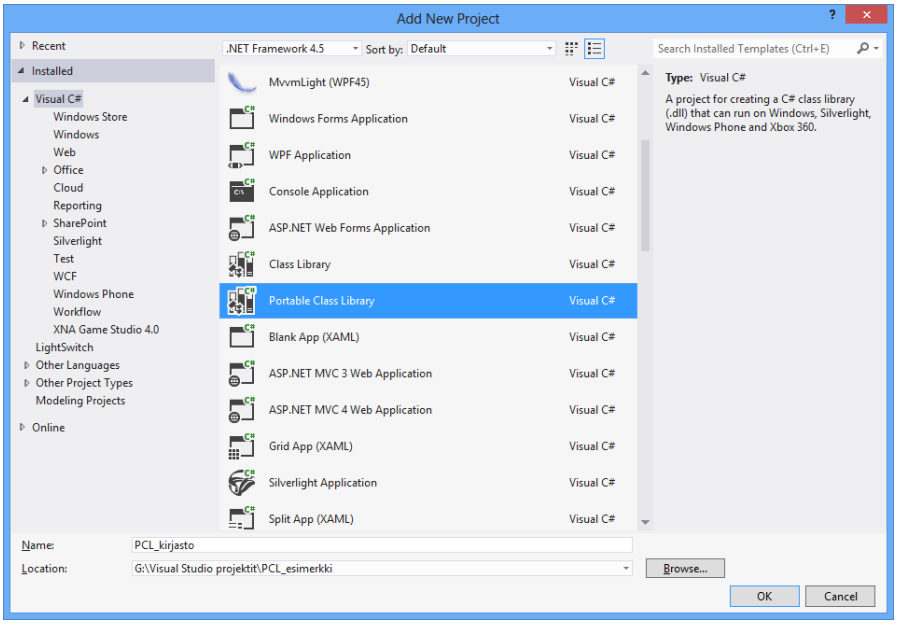
PCL tukee seuraavia alustoja: Windows 7, Windows 8, Windows Phone 7.x, Windows Phone 8, Silverlight ja Xbox 360. Tarvittavat alustat voi valita projektia luodessa. Jos otetaan käyttöön ainoastaan muutama alusta, saadaan käyttöön isompi määrä tuettuja kirjastoja. Jos otetaan käyttöön PCL, joka tukee Xbox 360-alustaa, ei voida jakaa muiden alustojen kanssa kuin System.Core- ja System.XLINQ -kirjasto. Käyttöliittymä koodia ei voi jakaa PCL:ssä, koska kaikkien alustoiden käyttöliittymäkirjastot eroavat hieman toisistaan. Taulukossa 1 on esitelty yhteensopivat kirjastot. (Trofin, M. 2012; MSDN. Cross-Platform Development with the .NET Framework. n.d.)

Taulukko 1 Tuetut ominaisuudet eri alustoilla PCL:ssä Lähde: MSDN. Cross-Platform Development with the .NET Framework. n.d.

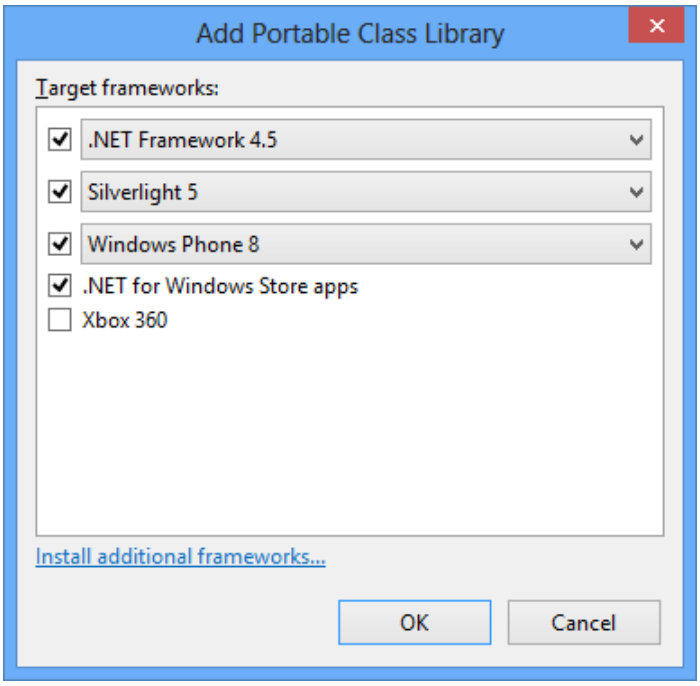
Ominaisuus	**.NET**	**Windows Store**	**Silver-light**	**Windows Phone**	**Xbox 360**
Core	x	x	x	x	x
LINQ	x	x	x	x	
IQueryable	x	x	x	7.5<	
Dynamic keyword	4.5	x	x		
Managed Extensibility Framework (MEF)	x	x	x		
Network Class Library (NCL)	x	x	x	x	
Serialization	x	x	x	x	
Windows Communica-tion Foundation(WCF)	x	x	x	x	
Model-View-View Model (MVVM)	4.5	x	x	x	
Data annotations	4.03 & 4.5	x	x		
XLINQ	4.03 & 4.5	x	x	x	x
System.Numerics	4.03 & 4.5	x	x		

3.2 Portable Class Libraryn käyttö

PCL:n lisääminen olemassa olevaan projektiin on hyvin yksinkertaista. Ei tarvitse kuin ainoastaan lisätä uusi Portable Class Library projekti Visual Studion olemassa olevaan solutioniin (Kuva 3). Tämän jälkeen aukeaa ikkuna, josta voi valita tuetut alustat (Kuva 4). Tuettuja alustoja voi muokata jälkikäteen projektin asetuksista, jonka jälkeen kääntäjä käy läpi uudet valinnat ja valitsee yhteensopivat luokat automaattisesti.

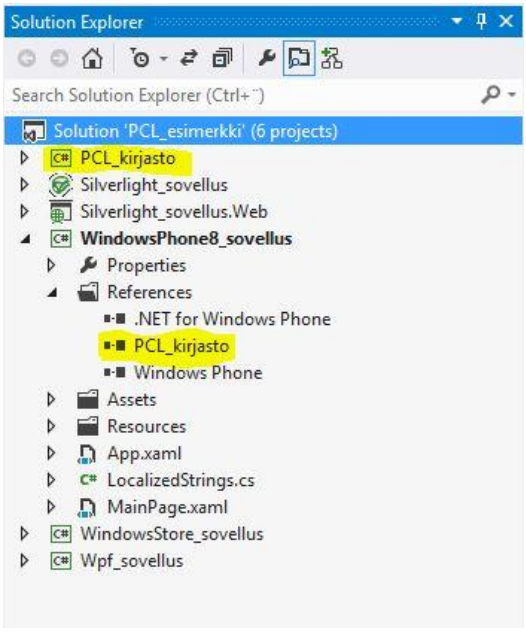


Kuva 3 PCL:n lisääminen Visual Studion solutioniin



Kuva 4 Tuettavien alustojen valinta

Kun PCL-projekti on lisätty solutioniin, tehdään muista projekteista viittaus (*reference*) PCL-projektiin. Tämän jälkeen kaikki projektit voivat käyttää samaa koodia PCL:stä. Kuvassa 5 on esimerkki projektirakenteesta. Kuten esimerkistä näkee, solutioniin on lisätty WPF-, Windows Phone 8-, Windows Store- ja Silverlight-projektit. Jokainen projekti voi käyttää PCL:ää, jossa voitaisiin pitää muun muassa yhteisiä malliluokkia, WCF-kutsuja, HTTP-kutsuja ja XML-tiedostojen käyttö. Näin ollen koodin hallittavuus paranee merkittävästi, kun ei ole tarvetta kopioida jokaiseen projektiin samaa koodia uudelleen.



Kuva 5 Esimerkki projektirakenteesta

3.3 Rajapinnat ja abstraktit luokat

Kun PCL:ssä on tarve käyttää sellaista luokkaa, mitä se ei tue, silloin pitää ottaa mukaan joko abstrakti luokka tai rajapinnat. Tällöin voidaan toteuttaa PCL:ssä esimerkiksi abstrakti luokka, johon määritellään halutut kentät (*fields*), ominaisuudet (*properties*), metodit (*methods*) ja tapahtumat (*events*). (MSDN. n.d. Cross-Platform Development with the .NET Framework.)

Esimerkissä 11 on toteutettu abstraktiluokka, jonka tarkoituksena on tallentaa käyttäjän nimi. Koska jokaisella alustalla on erilaiset toteutukset tallennukseen ja tallennusta halutaan käyttää PCL:stä käsin, jokainen alusta voi toteuttaa luokan erikseen.

```
namespace PCL_kirjasto
{
    public abstract class AbstractLocalStorage
    {

        public static AbstractLocalStorage Toteutus
        {
            get;
            set;
        }

        public abstract void Tallenna(string nimi);
    }
}
```

Esimerkki 11 Abstraktin luokan määritys PCL:ssä

Jokainen alusta antaa oman toteutuksensa staattiselle Toteutus-ominaisuudelle. Esimerkissä 12 on toteutettu PCL:n AbstractLocalStorage-luokka. Esimerkissä 13 annetaan käynnistyksen yhteydessä AbstractLocalStorage-luokan staattiselle Toteutus-ominaisuudelle uusi instanssi alustakohtaisesta toteutuksesta. Ei ole pakollista antaa toteutusta käynnistyksen yhteydessä, sen voi myös tehdä tarpeen mukaan myös myöhemminkin.

```
using Windows.Storage;
using PCL_kirjasto;

namespace WindowsStore_sovellus
{
    public class LocalStorage : AbstractLocalStorage
    {

        public override void Tallenna(string nimi)
        {
            ApplicationData.Current.LocalSettings.
            Values["asetukset"] = nimi;
        }
    }
}
```

Esimerkki 12 Abstraktin luokan toteutus Windows Store -sovelluksessa

```
protected override void OnLaunched(LaunchActivatedEventArgs args)
{
    AbstractLocalStorage.Toteutus = new LocalStorage();
}
```

Esimerkki 13 Sovelluskohtaisen toteutuksen antaminen käynnistyksen yhteydessä

4 WINDOWS RUNTIME COMPONENT

Windows Runtime Component on Windows 8:n mukana tullut ominaisuus, joka löytyy myös Windows Phone 8:sta nimellä Windows Phone Runtime Component. Ne ovat kieliriippumattomia komponentteja, mikä mahdollistaa esimerkiksi C++:lla tehdyn komponentin kutsumisen C#:lla kirjoitetusta Windows 8- ja Windows Phone 8 -sovelluksesta. (MSDN, 2013. Share Using Windows Runtime Components.)

Windows Runtime Component on hieman samanlainen kuin perinteinen .dll tiedosto. Käännettynä siitä ei kuitenkaan tule perinteistä .dll tiedostoa vaan .winmd tiedosto, joka sisältää koodin lisäksi tarvittavan metadatan, jonka avulla sitä voi käyttää muilla kielillä. Windows Runtime ja Windows Phone Runtime perustuu kokonaan kyseisiin komponentteihin. (MSDN. 2013. Share Using Windows Runtime Components; Lovell, M. 2011.)

4.1 Milloin kannattaa käyttää Windows Runtime Componentia?

Windows Runtime Component on todella hyödyllinen, jos kyseessä on esimerkiksi sovellus, jossa tehdään aikaa vieviä laskutoimituksia. Tällöin laskutoimitukset voi toteuttaa C++:aa hyödyntäen, jolloin suorituskyky kasvaa. Komponenttia voi myös hyödyntää, jos tiimissä on esimerkiksi Visual Basic ohjelmoija ja suunniteltua sovellusta ollaan tekemässä C#:lla. Komponenttia voi myös käyttää, jos halutaan toteuttaa toiminto, jota voidaan hyödyntää helposti muissakin sovelluksissa. Tämä toimii myös toisinpäin. Jos halutaan käyttää kolmannen osapuolen kirjastoja, voidaan kirjaston käyttö kääriä Windows Runtime Componenttiin ja käyttää sitä omasta sovelluksesta. Windows Runtime Componentin avulla voidaan myös käyttää tuettuja Win32, COM ja Direct3D rajapintoja. (MSDN, 2013. Share Using Windows Runtime Components.)

Windows Runtime Componentin käyttöön ja tekemiseen liittyy myös rajoitteita. Taulukossa 2 on esitelty, millä ohjelmointikielellä voi toteuttaa Runtime Componenteja ja millä kielillä niitä voi kutsua sovelluksesta. Rajoitukset komponentin teosta löytyvät osoitteesta: http://msdn.microsoft.com/en-us/library/windows/apps/br230301.aspx.

Taulukko 2 Windows Runtime Componentin ja Windows Phone Runtime Componentin tuetut ohjelmointikielet

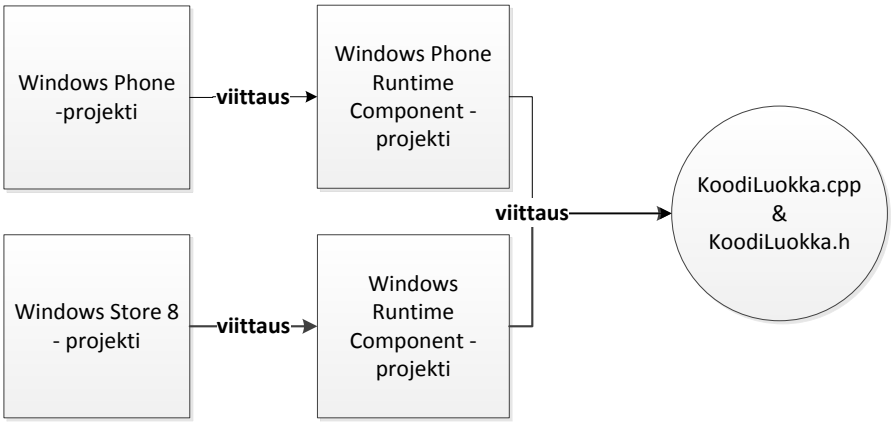
Alusta	Teko	Käyttö
Windows Phone 8	C++	C++, C#, Visual Basic
Windows 8	C++, C#, Visual Basic	C++, C#, Visual Basic, JavaScript

4.2 Windows Runtime Componentin käyttö

Runtime Componenteissa käytetään Windows Runtime -tyyppejä .NET Framework -tyyppien sijasta. Kuitenkin jos komponenttia kutsutaan hallitulla koodilla (*managed code)* kirjoitetusta ohjelmasta, tästä ei tarvitse välittää. Visual Studion IntelliSense-toiminto kääntää Windows Runtime -tyypit vastaamaan .NET-tyyppejä. Komponenttia voi siis käyttää kuin käytettäisiin mitä tahansa kirjastoa. Jos komponentti kirjoitetaan hallitulla koodilla, se voidaan tehdä edelleen .NET-tyyppejä käyttäen. Komponenttia käännettäessä .NET-tyypit käännetään automaattisesti Runtime Metadata Export Toolilla Runtime-tyypeiksi. Natiivia koodia käytettäessä komponentissa ja kutsuvassa sovelluksessa, tällaista muunnosta ei tapahdu. (MSDN. n.d .NET Framework mappings of Windows Runtime types.)

Runtime Component luodaan Visual Studiossa. Projektin tyypiksi valitaan Windows Runtime Component Windows Store -sovellukselle ja Windows Phone Runtime Component Windows Phone 8 -sovellukselle. Tässä on syytä huomioida, että vaikka komponentissa voidaan käyttää samaa koodia tehdessä sovellusta Windows 8:lle ja Windows Phone 8:lle, ne tarvitsevat kuitenkin omat komponenttiprojektit kummallekin.

Kuvassa 6 on esimerkki, kuinka voidaan hyödyntää C++ koodia Runtime Componentien avulla tehtäessä sovellusta Phone 8:lle ja Windows 8:lle. Esimerkissä on kirjoitettu C++:lla luokka, jota halutaan käyttää kummastakin sovelluksesta käsin. Kummallekin on tehty oma Runtime Component -projekti, joista viitataan samoihin tiedostoihin. Viittaus toteutetaan kummallekin komponenttiprojektille erikseen Visual Studiossa käyttäen toimintoa Add->Existing Item ja valitaan halutut luokat. Näin tehtäessä projekti ei enää käänny, koska komponenttiprojektien nimiavaruudet eivät ole samanlaisia kuin viitatussa luokassa. Kummallakin komponenttiprojektilla on projektin ominaisuuksissa Root Namespace -asetus, joka pitää vaihtaa viitatun luokan kanssa samaksi. Esimerkiksi jos KoodiLuokka.cpp:n ja KoodiLuokka.h:n nimiavaruus olisi RuntimeComponent, pitäisi komponentti projektien juurinimiavaruudeksi määrittää sama RuntimeComponent.



Kuva 6 Esimerkki projektirakenteesta jaettaessa C++ koodia Runtime Componentien avulla Windows Phone 8- ja Windows Store 8 -sovelluksissa

Kuten aiemmin on mainittu, ei komponentin käyttö juurikaan eroa perinteisen kirjaston käytöstä. Muutamia poikkeuksia kuitenkin on johtuen Windows Runtime tyyppien käännöksestä .NET-tyypeiksi ja toisinpäin. Esimerkiksi tästä johtuen string-muuttuja ei voi olla null-arvoinen käännettäessä Runtime tyypistä .NET-tyypiksi vaan se on aina tyhjä. Jos taas Runtime-metodille annetaan .NET:n string-muuttuja, joka on arvoltaan null lopputuloksena on ArgumentNullException-virhe. Nämä luokat, joiden kanssa on syytä olla tarkkana Runtime Componentin ottaessa niitä parametriksi tai palauttaessa arvoa ovat String, DateTime, DateTimeOffset, Array, Uri, IReference<T> ja Nullable<T>.
(Windows Dev Center. n.d. The CLR and the Windows Runtime.)

Suurimmaksi osaksi kuitenkaan Runtime luokkien käännös .NET luokiksi ei näy komponentin käyttäjälle. Esimerkissä 14 on tehty C++:lla Windows Runtime Component -luokka.

```
#include "pch.h"
#include "VaativatLaskuToimitukset.h"
using namespace WindowsRuntimeComponent;
using namespace Platform;
using namespace Platform::Collections;
using namespace Windows::Foundation::Collections;

VaativatLaskuToimitukset::VaativatLaskuToimitukset()
{}
int VaativatLaskuToimitukset::LaskeSumma(int n1, int n2)
{
    int summa = n1 + n2;
    return summa;
}
IVector<double>^ VaativatLaskuToimitukset::AnnaLista()
{
    auto lista = ref new Vector<double>();
    lista->InsertAt(0,10.0);
    lista->InsertAt(1,20.0);
    return lista;
}
```

Esimerkki 14 VaativatLaskuToimitukset.cpp luokka

Esimerkkiluokassa on kaksi metodia, joista toinen ottaa kaksi int-tyyppistä parametria ja palauttaa niiden yhteenlasketun tuloksen int-muodossa. Toinen metodi ei ota yhtään parametria mutta palauttaa IVector-rajapinnan toteuttavan listan, missä on muutama double-arvo.

Esimerkeissä 15 ja 16 on käytetty samaa Windows Runtime Componentia mutta eri kielellä. C++:lla tehdystä esimerkistä on syytä huomata, että listan tulos otetaan IVector muuttujaan, kun taas C#:lla tehdystä esimerkistä nähdään, että IVector-rajapinnan palauttava metodi palauttaa IList-rajapinnan. Esimerkistä nähdään, kuinka Runtime-luokat kääntyvät automaattisesti .NET-luokiksi kutsuttaessa niitä hallitulla koodilla kirjoitetusta ohjelmasta.

```
auto l =
ref new WindowsRuntimeComponent::VaativatLaskuToimitukset();
int summa = l->LaskeSumma(5,10);
IVector<double>^ lista = l->AnnaLista();
```

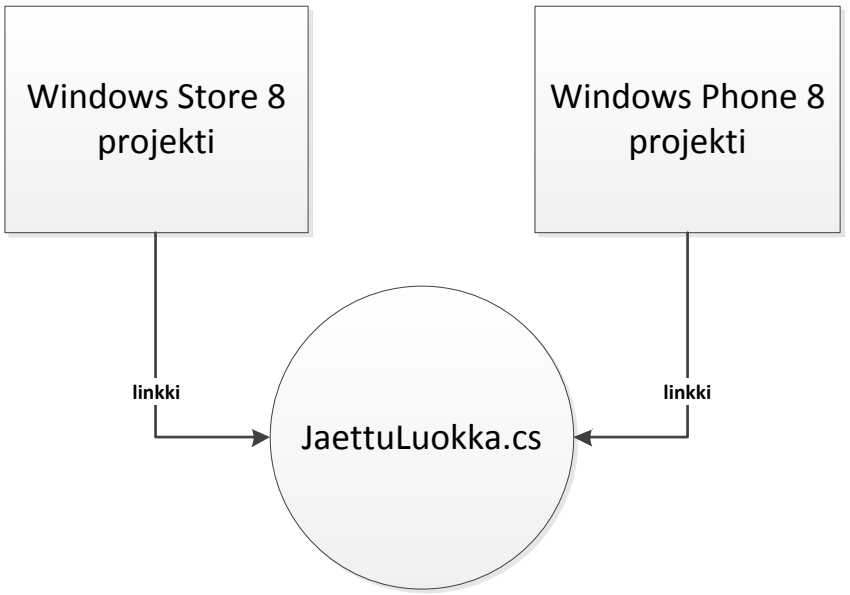
Esimerkki 15 Windows Runtime Componentissa olevan VaativatLaskuToimitukset-luokan käyttö C++ ohjelmassa

```
var l =
new WindowsRuntimeComponent.VaativatLaskuToimitukset();
int summa = l.LaskeSumma(1, 7);
IList<double> lista = l.AnnaLista();
```

Esimerkki 16 Windows Runtime Componentissa olevan VaativatLaskuToimitukset-luokan käyttö C# ohjelmassa

5 JAETTU KOODI

Jaetuksi koodiksi kutsutaan koodia, jota ei voi jakaa käyttämällä Portable Class Librarya, mutta se on silti keskenään yhteensopivaa tai jaettavaa on niin vähän, ettei ole syytä käyttää PCL:ää. Koodia jaetaan kirjoittamalla luokka yhteen kertaan ja jakamalla sitä linkkinä projekteissa. Projektia ajettaessa linkitetty tiedosto haetaan ja käännetään MSIL-muotoon kuten normaalikin luokka. Luokan kopiointiin nähden tässä kaikki luokkaan tehdyt muutokset päivittyvät kaikkiin projekteihin. Tämä helpottaa koodin ylläpitoa huomattavasti, kun ei tarvitse päivittää useaa tiedostoa. Ainoa ero normaaliin luokkaan projektissa on se, että Visual Studio näyttää tiedoston edessä pienen ikonin. Kuvassa 7 on havainnollistettu jaetun koodin käyttöä (Rothaus & Byrne. 2012.)



Kuva 7 Esimerkki jaetun luokan käytöstä Windows Store 8- ja Windows Phone 8 -projekteissa

5.1 Mitä voi jakaa linkkinä?

Linkitettynä tiedostona voi jakaa melkein kaikkea koodia. Koodin pitää kuitenkin olla yhteensopivaa siellä, mihin se linkitetään. .NET koodia kannattaa kuitenkin jakaa PCL:ssä niin paljon kuin on mahdollista. Kaikkea PCL ei kuitenkaan tue, tällöin voidaan PCL-yhteensopimaton koodi jakaa linkitettynä projektien välillä ja käyttää sitä rajapinnan tai abstraktin luokan avulla PCL:ssä. (MSDN. 2013. Share code with Add as Link.)

Windows 8 ja Windows Phone 8 -ohjelmien välillä voi myös jakaa UserControlleja, jotka on tehty XAMLia käyttäen. Windows 8 ja Windows Phone 8 eroavat toisistaan hieman XAMLin osalta. Eroavaisuudet pitää ottaa huomioon kontrolleja jakaessa. XAMLissa määritellyt nimiavaruudet ovat kummassakin osaltaan erilaiset. Ratkaisuna kontrollista poistetaan XAMLin puolelta erilaiset nimiavaruudet ja siirretään ne taustakooditiedostoon using-lauseeksi. Taustakooditiedostossa alustakohtaiset nimiavaruudet kääritään ehdollisiin kääntäjän vihjeisiin, jolloin kääntäjä huolehtii oikeista nimiavaruuksista. Esimerkissä 17 on toteutettu yhteinen UserControl. Jos esimerkin koodi ajetaan Windows 8 koneella, niin ehdolliset kääntäjän vihjeet käyttävät if-lohkossa olevia nimiavaruuksia. NETFX_CORE-merkintä on Windows Store sovelluksissa automaattisesti määritelty tarkoittamaan Store-sovellusta. (MSDN. 2013. Sharing XAML UI.)

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Net;

#if NETFX_CORE
using Windows.UI.Xaml.Controls;
using Windows.UI.Xaml;
#else
using System.Windows;
using System.Windows.Controls;
using System.Windows.Navigation;
using Microsoft.Phone.Controls;
using Microsoft.Phone.Shell;
#endif

namespace TestiSovellus.JaettuKoodi
{
    public partial class JaettuUserControl : UserControl
    {
            //Luokkakoodit
    }
}
```

Esimerkki 17 Jaettu UserControl, jossa kääntäjä huolehtii oikeista nimiavaruuksista käännöksen aikana

Windows Phone 8 ja Windows 8 jakavat osaltaan samaa Windows Runtime -rajapintaa. Tämän rajapinnan avulla voi käyttää muun muassa sensoreita, tiedostoon tallennusta, verkkoa ja monia muita Windows Runtime -luokkia. Silloin kun halutaan käyttää sellaista rajapintaa, jota kumpikin alusta tukee, se on hyödyllistä kirjoittaa linkitettyyn luokkaan. (MSDN. 2013. Windows Phone Runtime API.) Täydellinen listaus yhteisistä rajapinnoista löytyy osoitteesta:
http://msdn.microsoft.com/en-us/library/windowsphone/develop/jj207212(v=vs.105).aspx#BKMK_PhoneadditionstowinrtAPIadoptedfromwin_8_client

Vaikka osa rajapinnoista on yhteisiä, ei jokaista luokkaa välttämättä ole vielä toteutettu rajapinnan sisällä Windows Phone 8 -alustalle. MSDN:n sivustolla on jokaisessa luokassa erikseen maininta, jos kyseistä luokkaa ei ole toteutettuna puhelimelle. Myös luokkaa käytettäessä Visual Studion IntelliSense-toiminto näyttää, ettei kyseistä luokkaa ole toteutettu. (MSDN. 2013. Windows Phone Runtime API.)

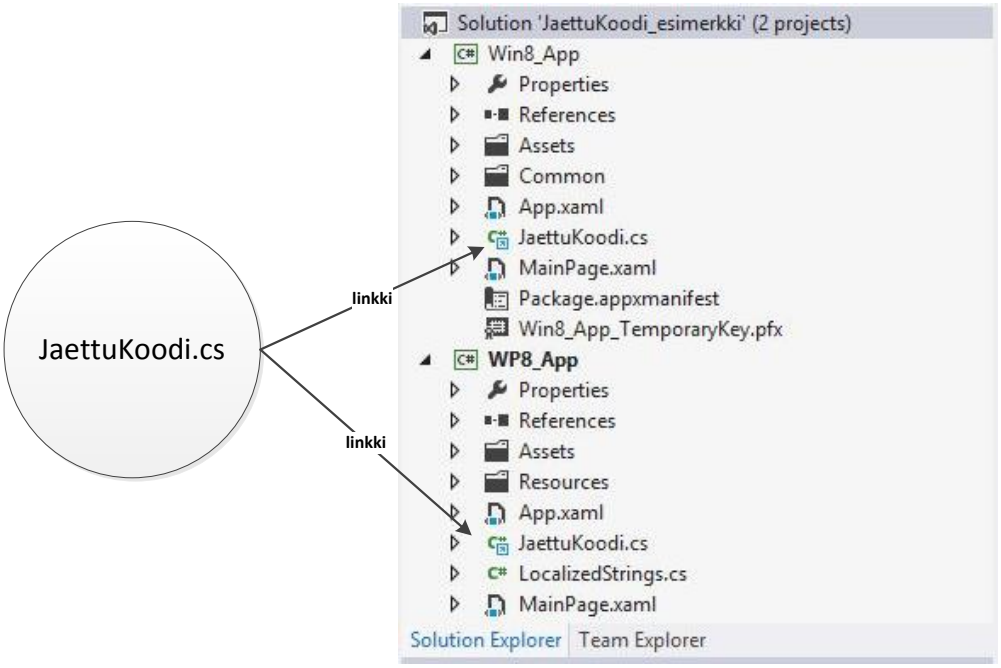
5.2 Jaetun koodin käyttö

Koodiluokan jakaminen linkittämällä on hyvin yksinkertainen toimenpide. Kirjoitetaan normaaliin tapaan luokka ja linkitetään se projekteihin. Esimerkissä 18 on yksinkertainen luokka, joka käyttää molemmille yhteistä Windows Runtime -rajapintaa.

```
public class JaettuKoodi
 {
        private Geolocator _gps;

        public JaettuKoodi()
        {
            _gps = new Geolocator();
        }

        public async Task<double[]> KayttajanCoordinaatit()
        {
            Geoposition sijainti =
            await _gps.GetGeopositionAsync();
            double[] koordinaatit = new double[2];

            koordinaatit[0] = sijainti.Coordinate.Latitude;
            koordinaatit[1] = sijainti.Coordinate.Longitude;

            return koordinaatit;
        }
  }
```

Esimerkki 18 JaettuKoodi-luokka, jossa käytetään yhteistä Windows Runtime -rajapintaa

Tämän luokan voi linkittää kumpaankin projektiin käyttämällä Visual Studion toimintoa Add As Link. Halutusta projektista valitaan toiminto Add->Existing Item ja valitaan linkitettävä luokka, minkä jälkeen valitaan Add-napin viereisestä pudotusvalikosta Add As Link. Tämän jälkeen luokka ilmestyy projektiin ja luokan edessä on pieni ikoni, josta ilmenee, että luokka on linkitetty. Kuvassa 8 on projektirakenne yhdellä jaetulla luokalla.



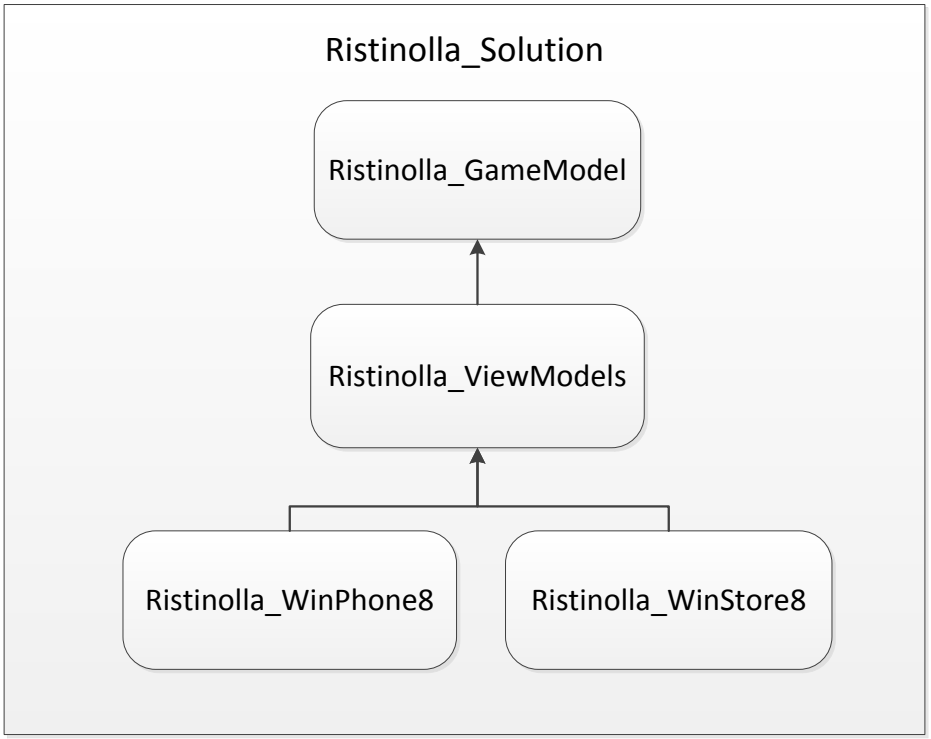
Kuva 8 Esimerkki solution-rakenteesta, jossa on linkitetty sama luokka Windows 8- ja Windows Phone 8 -projekteihin

6 KOODIN JAKAMINEN RISTINOLLA-PELISSÄ

Työn tavoitteena oli toteuttaa ristinolla-peli Windows Phone 8:lle ja Windows 8:lle. Peli on verkossa pelattava kaksinpeli jossa on myös globaali keskustelutoiminto. Pelissä voi valita halutun ruudukon koon kolmesta kymmeneen. Jos valittuna on 3 x 3 ruudukko, pelin voittamiseen vaaditaan kolme merkkiä vaakaan, pystysuoraan tai vinottain. 4 x 4 ja 5 x 5 ruudukossa voittoon vaaditaan neljä, ja sitä suuremmissa vaaditaan viisi. Peli päättyy, kun toinen pelaaja voittaa tai ruudukko on täysi. Pelin verkkototeutus on tehty Windows Communication Foundation tekniikalla ja palvelua isännöidään Azuren pilvipalvelussa. Pelin sovellusrakenne on tehty MVVM-mallia käyttäen, jonka toteutukseen käytettiin apuna MVVM Light Toolkit -kirjastoa.

Peli toteutettiin kolmen hengen ryhmässä. Työstä tehtiin kolme opinnäytetyötä tämä opinnäytetyö mukaan lukien. Niko Kuusisen työssä käsitellään Windows 8 ja Windows Phone 8 sovelluskehitystä yleisellä tasolla. Jarno Niemen työssä tutustutaan Windows Communication Foundationiin ja Microsoftin Azure pilvipalveluun.

Sovellusten projektit jaettiin neljään osaan: Ristinolla_GameModel-projektiin, joka toimii mallikerroksena, jossa on pelin varsinainen logiikka ja ruudukon tarkistus. Ristinolla_ViewModels-projektiin, jossa on näkymämallit, jotka kapseloivat mallin ja tarjoavat näkymälle toiminnollisuutta. Sitten on Windows Phone 8:lle ja Windows 8:lle omat projektinsa, jotka toimivat näkymäkerroksina. Kuvassa 9 on esitetty projektin rakenne ja osien riippuvuudet.



Kuva 9 Ristinolla-pelin projektirakenne

6.1 MVVM Light Toolkit -kirjasto

MVVM Light Toolkit on Laurent Bugnionin kehittämä ja ylläpitämä avoimen lähdekoodin projekti, jonka lähdekoodi on saatavilla CodePlex-palvelusta. Kirjoittamisen hetkellä uusin versio on 4.1.27.0 ja se on saatavana Nuget-pakettina Visual Studioon. Uusin .msi:nä saatava paketti on 4.1.24.0. Toolkitin tavoitteena on tarjota MVVM-mallin käyttöön apuja erilaisten apuluokkien avulla. (Bugnion, L. n.d. MVVM Light Toolkit; CodePlex n.d.)

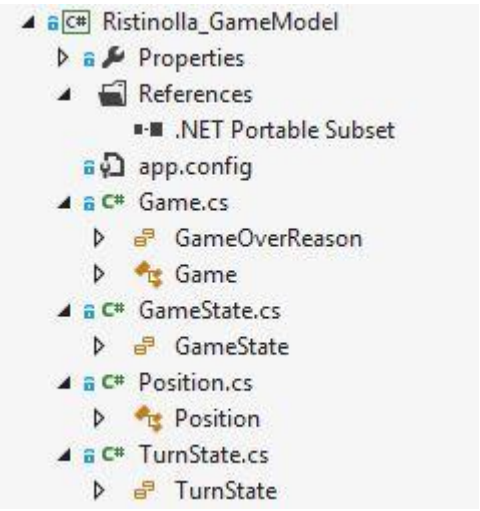
Toolkit tarjoaa monia hyödyllisiä kirjastoja ja apuluokkia. Taulukossa 3 on listaus yleisimmistä. Alustakohtaisesti on joitain lisäluokkia tarjolla. Kirjasto on hyvin yleisesti käytössä, mutta siitä ei löydy kovin hyviä ohjeita. Pääsääntöisesti löytyy pieniä blogikirjoituksia eri toiminnoista. Tästä syystä kannattaa käydä lähdekoodia läpi ja katsoa, mitä mikäkin luokka tekee. MVVM Light Toolkitin asennettavasta .msi paketista saa lisäksi käyttöönsä Visual Studioon projektimalleja ja valmiita koodipätkiä (*snippets*) jotka nopeuttavat koodin kirjoittamista huomattavasti.

Taulukko 3 MVVM Light Toolkitin yleisimmin käytetyt toiminnot. Lähde: Bugnion, L. n.d. MVVM Light Toolkit.

abstrakti ViewModelBase-luokka	Tästä luokasta voi periä näkymämallin. Luokassa on mm. INotifyProperyChanged-rajapinta toteutettu. Luokka tarjoaa monenlaisia metodeja PropertyChanged-tapahtuman laukaisuun. Luokka myös pitää Messenger-luokan instanssin sisällään
Messaging-nimiavaruus	Käytetään ohjelman sisällä viestimiseen. Esimerkiksi näkymämallit voivat tämän avulla ilmoittaa muutoksista toisilleen. Lähettäjän ei tarvitse tietää mitään vastaanottajasta kuin ei myöskään vastaanottajan lähettäjästä. Viesti voi sisältää monimutkaisiakin luokkia tai se voi olla ihan yksinkertainen arvo.
ObservableObject-luokka	Tämä luokka toteuttaa INotifyPropertyChanged-rajapinnan ja tarjoaa erilaisia metodeja PropertyChanged-tapahtuman laukaisuun. Luokka on tarkoitettu mallikerroksen luokissa käytettäväksi, jos niitä haluaa käyttää tiedonsidonnassa.
Command-nimiavaruus	Tarjoaa geneerisen ja geneerittömän RelayCommand-luokan, luokka toteuttaa ICommand-rajapinnan.
Extras-kirjaston	Kirjastossa on SimpleIoc-luokka, joka tarjoaa yksinkertaisen IServiceLocator-rajapinnan toteutuksen. Käytetään näkymien sitomiseen näkymämalleihin

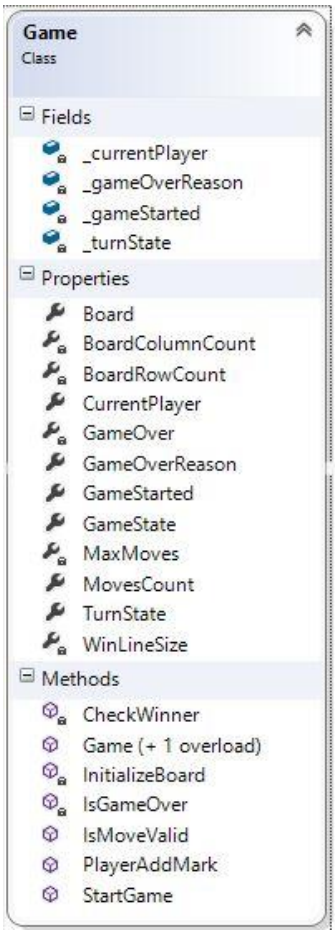
6.2 Mallikerros

Sovelluksen mallikerroksena on Ristinolla_GameModel-projekti. GameModel on PCL-kirjasto, joka tukee .NET Framework 4.5:sta, Silverlight 5:sta, Windows Phone 8:a ja Windows Store .NET:a. Sinne on toteutettu pelin logiikka. Mallikerros ei ota kantaa verkkototeutukseen, eikä sillä ole riippuvuuksia muihin sovelluksen osiin. Mallikerroksen tehtävänä on ylläpitää peliruudukkoa, tarkistaa sitä ja pitää huoli vuoronvaihdosta. Periaatteessa peliä voi tämän avulla pelata vaikka konsolisovelluksella. Kuvassa 10 on esitetty mallikerroksen projektirakenne.



Kuva 10 Malli kerros Visual Studiossa

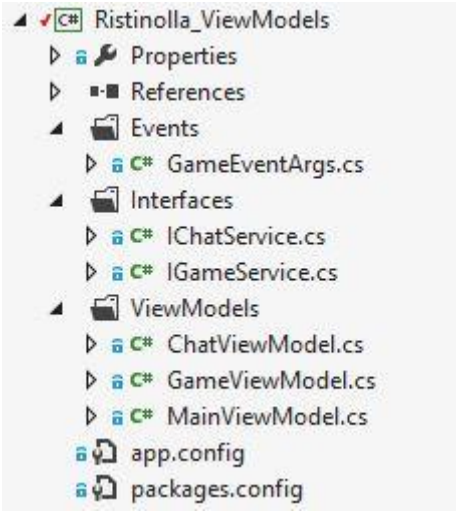
Kuten kuvasta 10 näkee, mallikerros on hyvin yksinkertainen. Siellä on kolme enum-rakennetta, jotka kuvastavat vuoroa, pelin tilaa ja sitä, miksi peli päättyi. Sitten on Position-luokka, jossa on ominaisuuksina rivi ja sarake int-muuttttujat, joiden avulla annetaan pelille siirtoja. Pelin logiikka löytyy Game-luokasta, jonka luokkakaavio on esitetty kuvassa 11.



Kuva 11 Game-luokan luokkakaavio

6.3 Näkymämallikerros

Näkymämallikerros on kaikista isoin kerros. Sillä on riippuvuus mallikerrokseen, ja sen tehtävä on kapseloida malli näkymiltä. Näkymämallikerros ei tiedä, kuka sitä käyttää. Kerros tarjoaa näkymille sidottavia ominaisuuksia, komentoja ja tiedon tarkistusta. Esimerkiksi mallikerros ei välitä, kuka asettaa merkin, onko se ensimmäinen vai toinen pelaaja. Malli ainoastaan vaihtelee vuoroa joka merkin laiton jälkeen. Näkymämalli huolehtii siitä, ettei sama pelaaja pelaa toisen pelaajan vuorolla. Tämän se tekee asettamalla verkon päässä olevan pelaajan toiseksi pelaajaksi ja estää komennon suorittamisen silloin, kun ei ole oma vuoro. Näkymämalli-projektin rakenne on esitetty kuvassa 12.



Kuva 12 Näkymämallikerros Visual Studiossa

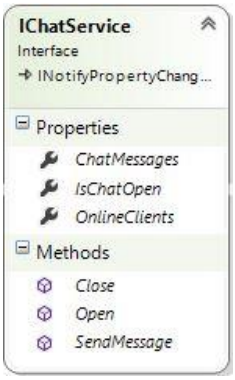
Projekti on oma PCL-kirjasto, jossa on tuettuna samat alustat kuin mallissakin. Projektiin on lisäksi asennettu seuraavat Nuget-paketit: MVVM Light Libraries only (PCL) ja Async for .NET Framework. Näillä paketeilla on saatu käyttöön MVVM Lightin apuluokat PCL kirjastoon ja uudet async ja await -avainsanat. Uuden .NET 4.5 version mukana tuli nämä kaksi uutta avainsanaa, joiden avulla voidaan toteuttaa asynkronisia toimintoja todella helposti.

Näkymämalleja on toteutettu kolme: pelille, keskustelulle ja sitten päävalikoille yms. Jokainen näkymämalli periytyy MVVM Light Toolkitin tarjoamasta ViewModelBase-luokasta, josta löytyy valmiina kaikkien näkymämallien tarvitsema INotifyPropertyChanged-rajapinnan toteutus. Näkymämalleissa käytetään komentoihin saman Toolkitin tarjoamaa RelayCommand-luokkaa. Näkymämallien väliseen keskusteluun käytetään Toolkitin Messenger-luokkaa, jolloin näkymämallitkaan eivät tiedä toisistaan mitään.

Näkymämallikerroksessa on esitelty kaksi rajapintaa: IChatService ja IGameService, jotka tarjoavat rajapinnan pelin verkkototeutukselle. Verkkotekniikat ovat yleensä alustakohtaisia, joten PCL-kirjastot eivät tue niitä, ja Windows Phone 8- ja Windows 8 -sovelluksissa on käytetty tässä työssä erilaisia toteutuksia. Windows 8 -sovelluksessa on käytetty WCF:n NetHttpBinding-tekniikkaa, mutta Windows Phone 8 -sovelluksessa tämä ei ole tuettuna, joten siinä on käytetty WebSocketeja, joihin edellä mainittu NetHttpBinding:kin perustuu. Rajapintaa käytetään GameViewModel- ja ChatViewModel -luokista, joilla ne keskustelevat verkon yli. Rajapinnat toteutetaan Windows Phone 8- ja Windows 8 -projektissa. Rajapinnan toteutus annetaan luokille niiden rakentajassa, kuten esimerkissä 19 on näytetty. Kuvassa 13 on IChatServicen luokkakaavio.

```
private IChatService _chatService;
public ChatViewModel(IChatService service)
 {
     _chatService = service;
 }
```

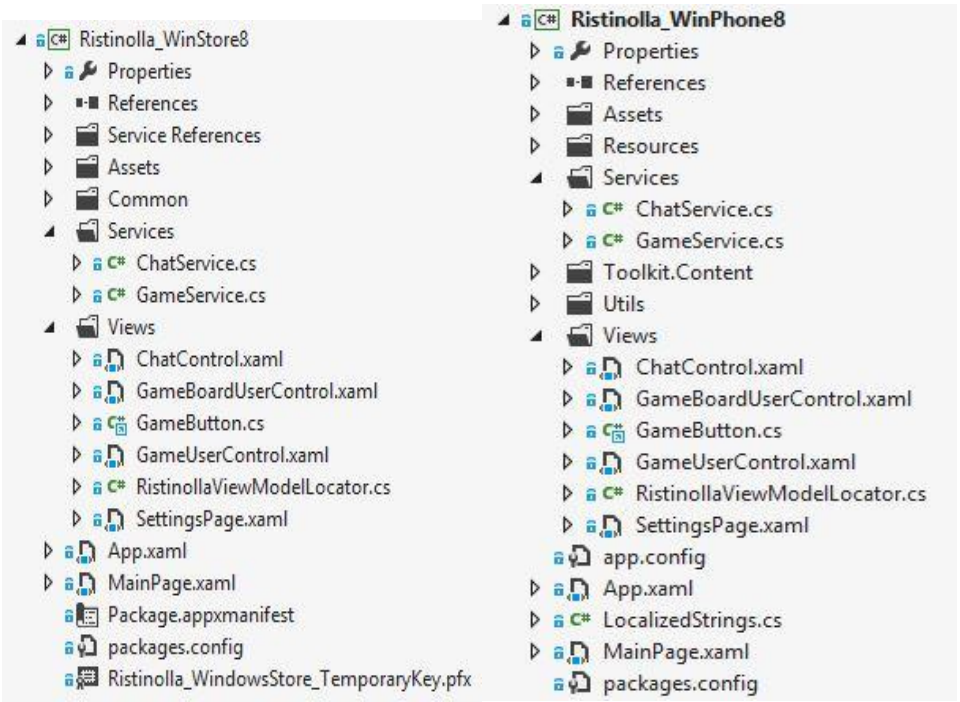
Esimerkki 19 ChatViewModelin rakentajassa IChatServicen asettaminen



Kuva 13 IChatService-rajapinnan luokkakaavio

6.4 Näkymäkerros

Näkymäkerroksia on toteutettu kaksi kappaletta, Windows Phone 8:lle ja Windows 8:lle omansa. Näiden kerrosten tehtävänä on tarjota käyttöliittymä pelille ja toteuttaa alustakohtaiset rajapinnat. Kumpaankin projektiin on lisätty Nuget-pakettina MVVM Light Libraries only (PCL). Windows Phone 8 -projektille on lisätty lisäksi Windows Phone Toolkit ja WebSocket4Net. Kuvassa 14 on esitetty projektien rakenteet.



Kuva 14 Windows Store- ja Windows Phone 8 -projektien rakenne

Solutionin kansion juureen on lisätty SharedCode-kansio, jossa on kaksi jaettua tiedostoa: GameButton.cs ja LocalStorage.cs. GameButton on Button-luokasta peritty kontrolli, johon on lisätty tarvittavaa toiminnollisuutta. Koska kummallekin toteutettuna luokka oli lähes identtinen, se päätettiin jakaa kummallekin linkkinä. Kuvasta 14 näkee, kuinka GameButton.cs tiedoston edessä on pieni sininen ikoni kummassakin projektissa. Yksi ero alustoiden välillä oli nimiavaruudet, esimerkiksi Storessa kontrollit sijaitsevat Windows.UI.Xaml.Controls nimiavaruudessa, kun Phonessa ne sijaitsevat System.Windows.Controls nimiavaruudessa. Yksi ero oli Button-luokasta ylikirjoitettavan OnApplyTemplate-metodin näkyvyysmääre. Storessa se on julkinen ja Phonessa suojattu. Nämä pienet erot korjattiin käyttämällä ehdollisia kääntäjän vihjeitä. Esimerkistä 20 näkyy, kuinka erot on korjattu niin, että luokka voitiin jakaa linkkinä kummallekin. Jos koodi ajetaan Windows Store -sovelluksella, kääntäjä kääntää if-lohkon sisällä olevat ja jättää huomiotta else-lohkossa olevat.

```
#if NETFX_CORE
using Windows.UI.Xaml;
using Windows.UI.Xaml.Controls;
#else
using System.Windows;
using System.Windows.Controls;
#endif
using Ristinolla_ViewModels.ViewModels;

    //..Koodia poistettu..
public class GameButton : Button
    {
        //....

#if NETFX_CORE
        protected override void OnApplyTemplate()
#else
        public override void OnApplyTemplate()
#endif
        {
            base.OnApplyTemplate();
            UpdateButtonLayout(false);
        }
         //....
```

Esimerkki 20 GameButton-luokan alustakohtaisten erojen korjaus

Yksi jaettu tiedosto sovellusten välillä oli LocalStorage.cs. Tiedosto tarjoaa User- ja LocalStorage -luokan. LocalStorage-luokassa on metodit asetusten ja käyttäjänimen tallennukseen fyysiselle laitteelle. Tämä tiedosto ei vaatinut ollenkaan kääntäjän vihjeitä, koska siinä käytettiin kummallekin yhteistä Windows Runtime Apia. Alun perin oli tarkoitus tallentaa asetukset ApplicationData.LocalSettings-ominaisuuteen, koska kyseinen ominaisuus on yhteisessä API:ssa. Mutta kuitenkaan tämä ominaisuus ei tarkemman tutkimuksen jälkeen ollut vielä toteutettu Windows Phone 8:lle. Näin ollen päädyttiin käyttämään asetusten tallentamista tiedostoon.

Näkymien sitomiseen näkymämalleihin työssä käytettiin MVVM Light Toolkitin tarjoamaa IServiceLocator-rajapinnan toteutusta. Työssä luokan nimi on RistinollaViewModelLocator. Luokan rakentajassa rekisteröidään näkymämallit SimpleIoc-luokalle. Tämän jälkeen jokaiselle näkymämallille luodaan get-ominaisuus, joka palauttaa tietyn näkymämallin. Kyseinen ominaisuus palauttaa aina saman näkymämallin instanssin. Eli näkymämallit ovat näin käytettynä singleton-luokkia. Saman luokan rakentajassa rekisteröidään myös näkymämallikerroksen rajapinnoille toteutukset. Esimerkissä 21 on Windows Phone 8 projektin RistinollaViewModelLocator-luokan rakentaja.

```
static RistinollaViewModelLocator()
 {
    ServiceLocator.SetLocatorProvider(() => SimpleIoc.Default);

    if (ViewModelBase.IsInDesignModeStatic)
    {}
    else
    {
      SimpleIoc.Default.Register<IChatService,ChatService>();
      SimpleIoc.Default.Register<IGameService,GameService>();
    }

    SimpleIoc.Default.Register<MainViewModel>();
    SimpleIoc.Default.Register<GameViewModel>();
    SimpleIoc.Default.Register<ChatViewModel>();
 }
```

Esimerkki 21 RistinollaViewModelLocator-luokan rakentaja

Esimerkissä asetetaan ensin Microsoftin tarjoamalle ServiceLocator-luokalle MVVM Light Toolkitin IServiceLocator-rajapinnan toteuttava SimpleIoc-luokan instanssi. Tämän jälkeen rekisteröidään näkymämallin rajapinnoille alustakohtaiset toteutukset. Tämän jälkeen rekisteröidään näkymämallit.

Esimerkissä 22 on MainViewModelin palauttava Main-ominaisuus. Tätä ominaisuutta kutsuttaessa ServiceLocator tarkistaa, onko luokkaa rekisteröity ja jos on, niin tarkistetaan, onko sitä luotu. Jos ei, niin luodaan MainViewModelista uusi instanssi ja jos on, palautetaan jo olemassa oleva instanssi. Samanlainen ominaisuus löytyy jokaisesta näkymämallista erikseen.

```
   public MainViewModel Main
    {
      get
       {
          return
          ServiceLocator.Current.GetInstance<MainViewModel>();
       }
    }
```

Esimerkki 22 Näkymämallin tarjoavan ominaisuuden toteutus

RistinollaViewModelLocator-luokan instanssi alustetaan App.xaml tiedostossa, josta luokka on käytössä koko ohjelman ajan. Esimerkissä 23 on RistinollaViewModelLocatorin alustus App.xaml tiedostossa. Tämän jälkeen näkymä on helppo sitoa haluttuun näkymämalliin. Näkymän .xaml tiedostossa alustetaan DataContext-ominaisuus RistinollaViewModelLocatorin halutulla ominaisuudella ja lähteeksi merkitään App.xaml tiedostossa määritelty instanssi.

App.xaml:

```
<Application
    x:Class="Ristinolla_WinPhone.App"
    xmlns:view="clr-namespace:Ristinolla_WinPhone.Views">

    <!-- ..Koodia poistettu.. -->
<Application.Resources>
    <view:RistinollaViewModelLocator
            x:Key="Locator" d:IsDataSource="True"/>
<Application.Resources>
```

MainPage.xaml:

```
<phone:PhoneApplicationPage
    x:Class="Ristinolla_WinPhone.MainPage"
    DataContext="{Binding Main,
    Source={StaticResource Locator}}">
```

Esimerkki 23 RistinollaViewModelLocatorin alustus ja MainPage.xamlissa näkymämallin sitominen

6.5 Jatkokehitys

Jatkokehitystä ajatellen, työssä on hyvin toimiva pohja. Näkymämallit kuitenkin paisuvat hyvin herkästi, koska siellä on myös verkkorajapinnasta huolehtiminen. Tästä syystä sen voisi siirtää mallikerroksen hoitoon. Näkymien UserControlit voisi myös jakaa sovellusten kesken, kunhan otetaan käyttöliittymää tehtäessä huolellisesti erilaiset näyttöjen koot huomioon. Jos UserControleista tulee monimutkaisempia, niitä ei enää kannata jakaa. Helpommalla pääsee, kun tekee kummallekin omat, niin saa parhaan hyödyn käytetystä alustasta.

Pelin logiikan kasvaessa ja monimutkaistuessa, sen voisi kirjoittaa Windows Runtime Componentiin C++ kieltä käyttäen. Tämä taas aiheuttaisi hieman ongelmia, koska PCL:stä ei voi käyttää suoraan Runtime komponentteja, vaan toteutus pitäisi viedä sinne rajapintojen kautta. Tämä on kuitenkin mahdollista ja raskaammissa sovelluksissa tämä parantaisi vasteaikaa huomattavasti.

7 YHTEENVETO

Työn tarkoitus oli tutkia ja soveltaa ohjelmakoodin jakamisen tekniikoita. Tekniikoita ei ole kovin monia. Kuitenkin jokainen tekniikka tarjoaa hyvin erilaisen tavan käsitellä ohjelmakoodia alustasopivaksi. Nämä tekniikat yhdessä ja oikein käytettyinä mahdollistavat todella laajan yhteisen koodipohjan sovelluksille. Nämä tekniikat mahdollistavat myös tehokkaan yksikkötestauksen. Tämä aihe kuitenkin jätettiin työn ulkopuolelle.

Kaikista työssä esitellyistä tekniikoista PCL ja MVVM mahdollistavat suurimman jakohyödyn. PCL on todella hyvä keino rakentaa yhteensopivia kirjastoja sovellusten välille. MVVM-mallia käyttäen on mahdollista erottaa sovelluslogiikka pois näkymiltä, jolloin sovelluslogiikka voidaan siirtää PCL-projektiin. Tämän jaon jälkeen erikseen ei jää kuin käyttöliittymä ja tarvittavia sovelluskohtaisia tekniikoita kuten esim. verkon käyttö.

Suurin haaste jakotekniikoiden käyttöönotossa on MVVM-mallin opettelu. Vaikka sen käyttöönottoon on saatavilla monenlaisia kirjastoja mallin ymmärtäminen voi viedä aikansa. Mallin käyttöönoton perusedellytys on tiedonsidonnan hyvä ymmärrys. Tiedonsidonta on MVVM-mallin avaintekijä, jota ilman mallia ei voi käyttää. MVVM-mallissa kuin PCL:ssä polymorfismin käyttäminen on lähes pakollista. Näin saadaan käytettyä rajapintojen avulla sellaista koodia mikä pitää toteuttaa sovelluskohtaisesti.

Työn suunnittelu ja saattaminen sellaiseen pisteeseen jossa ohjelmakoodia voidaan jakaa saattaa kestää odotettua pidempään. Suunnittelu voi myös vaatia tarkempaa huomiota. Mutta ohjelman kasvaessa nämä tekniikat alkavat helpottaa huomattavasti sovelluskehitystä ja näin muutokset tulevaisuudessa eivät vaadi koko koodin uudelleenkirjoitusta.

LÄHTEET

Bugnion. L. n.d. MVVM Light Toolkit. Viitattu: 24.4.2013.
http://www.galasoft.ch/mvvm/

Bugnion, L. 2012. Using the MVVM Pattern in Windows 8. Viitattu 20.4.2013.
http://msdn.microsoft.com/en-us/magazine/jj651572.aspx

CodePlex. n.d. MVVM Light Toolkit. Viitattu: 24.4.2013.
http://mvvmlight.codeplex.com/

Garofalo, R. 2011. Building Enterprise Applications with Windows Presentation Foundation and the Model View ViewModel Pattern. United States: Microsoft Press.

Lovell, M. 2011. Lap around the Windows Runtime. Viitattu 15.4.2013
http://channel9.msdn.com/Events/BUILD/BUILD2011/PLAT-874T

MacDonald, M. 2010. Pro WPF in C# 2010. United States: Apress.

Mircea, T. 2012 Targeting Multiple Platforms with Portable Code: Overview. Viitattu 12.4.2013.
http://blogs.msdn.com/b/dotnet/archive/2012/07/06/targeting-multiple-platforms-with-portable-code-overview.aspx

MSDN,. n.d .NET Framework mappings of Windows Runtime types. Viitattu 18.4.2013.
http://msdn.microsoft.com/en-us/library/windows/apps/hh995050.aspx

MSDN, n.d. Cross-Platform Development with the .NET Framework.Viitattu 12.4.2013.
http://msdn.microsoft.com/en-us/library/gg597391.aspx

MSDN. n.d. Data Binding. Viitattu: 22.4.2013
http://msdn.microsoft.com/en-us/library/cc278072(v=vs.95).aspx

MSDN. n.d. Data Binding Overview. Viitattu 22.4.2013.
http://msdn.microsoft.com/en-us/library/ms752347.aspx

MSDN. 2013. Share code with Add as Link. Viitattu 19.4.2013
http://msdn.microsoft.com/en-us/library/windowsphone/develop/jj714082(v=vs.105).aspx

MSDN, 2013. Share Using Windows Runtime Components. Viitattu 15.4.2013.
http://msdn.microsoft.com/en-us/library/windowsphone/develop/jj714080(v=vs.105).aspx

MSDN . 2013. Sharing XAML UI. Viitattu : 19.4.2013.
http://msdn.microsoft.com/en-us/library/windowsphone/develop/jj714088(v=vs.105).aspx

MSDN. 2013. Windows Phone Runtime API. Viitattu 19.4.2013.
http://msdn.microsoft.com/en-us/library/windowsphone/develop/jj207212(v=vs.105).aspx#BKMK_PhoneadditionstowinrtAPIadoptedfromwin_8_client

Smith, J. n.d. Advanced MVVM. Josh Smith.

Rothaus, D, Byrne, A. 2012. How to Leverage your Code across WP8 and Windows 8 Viitattu 18.4.2013.
http://channel9.msdn.com/Events/Build/2012/3-043R

Windows Dev Center. n.d. The CLR and the Windows Runtime, docx-tiedosto Viitattu 18.4.2013
http://go.microsoft.com/fwlink/p/?LinkId=243099

Liite 1

Komento-luokan toteutus

```
using System;
using System.Windows.Input;

namespace MVVM_tietosidos_esimerkki
{
    public class Komento : ICommand
    {

        private readonly Action<object> _execute;
        private readonly Predicate<object> _canExecute;

        public Komento(Action<object> execute)
                       : this(execute, null)
        {
        }

        public Komento(Action<object> execute, Predicate<object> canExecute)
        {
            if (execute == null)
                throw new ArgumentNullException("execute");

            _execute = execute;
            _canExecute = canExecute;
        }

        public bool CanExecute(object parameter)
        {
            return _canExecute == null || _canExecute(parameter);
        }

        public event EventHandler CanExecuteChanged;

        public void RaiseCanExecuteChanged()
        {
            EventHandler handler = this.CanExecuteChanged;
            if (handler != null)
            {
                handler(this, new EventArgs());
            }
        }

        public void Execute(object parameter)
        {
            _execute(parameter);
        }
    }
}
```

Liite 2

Henkilo-luokan toteutus

```
namespace MVVM_esimerkki.Malli
{
    public class Henkilo
    {
        private string _etunimi;
        private string _sukunimi;

        public string Etunimi
        {
            get { return _etunimi; }
            set { _etunimi = value; }
        }

        public string Sukunimi
        {
            get { return _sukunimi; }
            set { _sukunimi = value; }
        }

        public void Tallenna()
        {
            //..Tänne voisi toteuttaa esim. tiedostoon kirjoituksen..
        }


    }
}
```

Liite 3

MainPageNäkymäMalli luokan toteutus

```
using System;
using System.Collections.ObjectModel;
using System.ComponentModel;
using System.Runtime.CompilerServices;
using MVVM_esimerkki.Malli;

namespace MVVM_esimerkki.NakymaMalli
{
    public class MainPageNäkymäMalli : INotifyPropertyChanged
    {

        public MainPageNäkymäMalli()
        {
            _henkilo=new Henkilo();
        }

        private Henkilo _henkilo;
        private ObservableCollection<Henkilo> _henkiloLista;

        private string _etunimi;
        private string _sukunimi;

        public string Etunimi
        {
            get { return _etunimi; }
            set
            {
                if(_etunimi==value)
                    return;

                _etunimi = value;
                RaisePropertyChanged("Etunimi");
                TallennaHenkiloKomento.RaiseCanExecuteChanged();
            }
        }


        public string Sukunimi
        {
            get { return _sukunimi; }
            set
            {
                if (_sukunimi == value)
                    return;

                _sukunimi = value;
                RaisePropertyChanged("Sukunimi");
                TallennaHenkiloKomento.RaiseCanExecuteChanged();
            }
        }

        public ObservableCollection<Henkilo> HenkiloLista
        {
            get
            {
                return _henkiloLista ??
                       (_henkiloLista = new ObservableCollection<Henkilo>());
```

```
            }
            set
            {
                if(_henkiloLista==value)
                    return;

                _henkiloLista = value;
                RaisePropertyChanged("HenkiloLista");
            }
        }

        private Komento _tallennaHenkilo;
        public Komento TallennaHenkiloKomento
        {
            get
            {
                return _tallennaHenkilo ??
                (_tallennaHenkilo =
                    new Komento(suorita =>
                                    {
                                        _henkilo.Etunimi = _etunimi;
                                        _henkilo.Sukunimi = _sukunimi;
                                        HenkiloLista.Add(_henkilo);
                                    },
                                voikoSuorittaa=>
                                    {
                                     if (!string.IsNullOrEmpty(_etunimi)
                                            &&
                                         !string.IsNullOrEmpty(_sukunimi))
                                             return true;

                                     return false;
                                    })); }
        }

        public event PropertyChangedEventHandler PropertyChanged;

        private void RaisePropertyChanged([CallerMemberName]
                                    String propertyName = "")
        {
            if (PropertyChanged != null)
            {
                PropertyChanged(this,
                                new PropertyChangedEventArgs(propertyName));
            }
        }
    }
}
```

MainPage.xaml ja MainPage.cs sivujen toteutus

MainPage.xaml:

```
<phone:PhoneApplicationPage
    x:Class="MVVM_esimerkki.Nakyma.MainPage"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:phone="clr-namespace:Microsoft.Phone.Controls;assembly=Micro
            soft.Phone"
    xmlns:shell="clr-namespace:Microsoft.Phone.Shell;assembly=Micro
            soft.Phone"
    xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
    xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"

    mc:Ignorable="d"
    FontFamily="{StaticResource PhoneFontFamilyNormal}"
    FontSize="{StaticResource PhoneFontSizeNormal}"
    Foreground="{StaticResource PhoneForegroundBrush}"
    SupportedOrientations="Portrait" Orientation="Portrait"
    shell:SystemTray.IsVisible="True">

    <Grid x:Name="LayoutRoot" Background="Transparent">
        <Grid.RowDefinitions>
            <RowDefinition Height="Auto"/>
            <RowDefinition Height="*"/>
        </Grid.RowDefinitions>


        <StackPanel x:Name="TitlePanel" Grid.Row="0" Margin="12,17,0,28">
            <TextBlock Text="MVVM esimerkki"
                       Style="{StaticResource PhoneTextNormalStyle}"
                       Margin="12,0"/>
            <TextBlock Text="etusivu" Margin="9,-7,0,0"
                       Style="{StaticResource PhoneTextTitle1Style}"/>
        </StackPanel>


        <Grid x:Name="ContentPanel" Grid.Row="1" Margin="12,0,12,0">
            <Grid.RowDefinitions>
                <RowDefinition Height="110"/>
                <RowDefinition Height="110"/>
                <RowDefinition Height="80"/>
                <RowDefinition Height="*"/>
            </Grid.RowDefinitions>

            <StackPanel Grid.Row="0">
                <TextBlock Text="Etunimi:" />
                <TextBox Text="{Binding Etunimi, Mode=TwoWay}"/>
            </StackPanel>
            <StackPanel Grid.Row="1">
                <TextBlock Text="Sukunimi:" />
                <TextBox Text="{Binding Sukunimi, Mode=TwoWay}" />
            </StackPanel>
            <Button Content="Tallenna"
                    Command="{Binding TallennaHenkiloKomento}"
                    Grid.Row="2"/>

            <Border Grid.Row="3" Width="300" CornerRadius="5"
```

```
                BorderBrush="Gray" BorderThickness="1">
            <ListBox ItemsSource="{Binding HenkiloLista}"
                     HorizontalContentAlignment="Stretch">
                <ListBox.ItemContainerStyle>
                    <Style TargetType="ListBoxItem">
                        <Setter Property="HorizontalContentAlignment"
                                Value="Stretch"/>
                    </Style>
                </ListBox.ItemContainerStyle>
                <ListBox.ItemTemplate>
                    <DataTemplate>
                        <Border BorderBrush="#FFFF3232"
                          BorderThickness="1" CornerRadius="4">
                        <StackPanel>
                            <StackPanel Orientation="Horizontal"
                                    Margin="3">
                                <TextBlock Text="Etunimi: "/>
                                <TextBlock Text="{Binding Etunimi}"
                                    Foreground="White"/>
                            </StackPanel>
                            <StackPanel Orientation="Horizontal"
                                    Margin="4,0,4,4">
                                <TextBlock Text="Sukunimi: "/>
                                <TextBlock Text="{Binding Sukunimi}"
                                            Foreground="White"/>
                            </StackPanel>
                        </StackPanel>
                        </Border>
                    </DataTemplate>
                </ListBox.ItemTemplate>
            </ListBox>
            </Border>
        </Grid>
    </Grid>

</phone:PhoneApplicationPage>
```

MainPage.cs:

```
using MVVM_esimerkki.NakymaMalli;
using Microsoft.Phone.Controls;

namespace MVVM_esimerkki.Nakyma
{
    public partial class MainPage : PhoneApplicationPage
    {
        public MainPage()
        {
            InitializeComponent();

            this.DataContext = new MainPageNäkymäMalli();
        }
    }
}
```